



# SMART CONTRACT AUDIT REPORT

for

CarbonVortex



Prepared By: Xiaomi Huang

PeckShield  
May 29, 2024

## Document Properties

Client	Bancor
Title	Smart Contract Audit Report
Target	CarbonVortex
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	May 29, 2024	Xuxian Jiang	Final Release
1.0-rc2	May 27, 2024	Xuxian Jiang	Release Candidate #2
1.0-rc1	May 18, 2024	Xuxian Jiang	Release Candidate #1

## Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About CarbonVortex . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Improved Precision in CarbonVortex::expectedTradeInput() . . . . .	11
3.2	Suggested Adherence Of Checks-Effects-Interactions Pattern . . . . .	12
3.3	Trust Issue of Admin Keys . . . . .	14
<b>4</b>	<b>Conclusion</b>	<b>17</b>
	<b>References</b>	<b>18</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `CarbonVortex` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About CarbonVortex

The `Carbon` protocol is a next-generation AMM which introduces the key new feature of asymmetric liquidity, wherein any given bonding curve only trades in a single direction, effectively being an out-of-the-money limit order. `CarbonVortex` collects fees from trades performed on the `Carbon DeFi` network, and as a result of arbitrage transactions performed using the `Bancor Fastlane`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of CarbonVortex

Item	Description
Issuer	Bancor
Website	<a href="https://www.carbondefi.xyz/">https://www.carbondefi.xyz/</a>
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 29, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers specifically the `CarbonVortex` contract.

- <https://github.com/bancorprotocol/carbon-contracts.git> (d06152e)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/bancorprotocol/carbon-contracts.git> (64d6dfc)

## 1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email ([contact@peckshield.com](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit




Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
<b>Error Conditions, Return Values, Status Codes</b>	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.



## 2 | Findings

### 2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the CarbonVortex protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	1	
Informational	1	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Precision in CarbonVortex::expectedTradeInput()	Coding Practices	Resolved
PVE-002	Informational	Suggested Adherence of Checks-Effects-Interactions	Coding Practices	Resolved
PVE-003	Medium	Trust on Admin Keys	Security Features	Mitigated

Beside the identified issues, we note that the staking support assumes the staked tokens are not deflationary. Also, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

## 3 | Detailed Results

### 3.1 Improved Precision in CarbonVortex::expectedTradeInput()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CarbonVortex
- Category: Numeric Errors [6]
- CWE subcategory: CWE-190 [2]

#### Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the the preferred handling when calculating the expected trade input amount.

In particular, we show below the implementation of the `CarbonVortex::expectedTradeInput()` routine. It has a rather straightforward logic in calculating the trade input based on the current price. However, the resulting amount is computed via the `MathEx.mulDivF()` helper (line 571), which is in favor of the trading user. We suggest the calculation should be computed in favor of the protocol with `MathEx.mulDivC()`. Though the resulting precision loss may be just a small number, it might play a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

```

562     function expectedTradeInput(Token token, uint128 targetAmount) public view
        validToken(token) returns (uint128) {
563         // revert if not enough amount available for trade
564         if (targetAmount > _amountAvailableForTrading(token)) {
565             revert InsufficientAmountForTrading();
566         }
567         Price memory currentPrice = tokenPrice(token);
568         // revert if current price is not valid

```

```
569     _validPrice(currentPrice);
570     // calculate the trade input based on the current price
571     return MathEx.mulDivF(currentPrice.sourceAmount, targetAmount, currentPrice.
        targetAmount).toUint128();
572 }
```

Listing 3.1: CarbonVortex::expectedTradeInput()

**Recommendation** Revise the above calculations in favor of the protocol when facing a possible precision loss.

**Status** The issue has been resolved by the following PR: 146.

## 3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: CarbonVortex
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

### Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [11] exploit, and the Uniswap/Lendf.Me hack [10].

We notice there is an occasion where the checks-effects-interactions principle is violated. Using the CarbonPOL as an example, the `_sellETHForBNT()` function (see the code snippet below) is provided to externally call `msg.sender` to refund any excess native token to caller (if the target token is native). However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. Apparently, the interaction with the external contract (line 630) starts before effecting the update on internal states (lines 638 and 643), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be abused of launching re-entrancy via the same entry function. Fortunately, the use of `nonReentrant` effectively blocks this specific risk.

```

614     function _sellTokenForTargetToken(Token token, uint128 targetAmount) private returns
        (uint128) {
615         uint128 sourceAmount = expectedTradeInput(token, targetAmount);
616         // revert if trade requires 0 target token
617         if (sourceAmount == 0) {
618             revert InvalidTrade();
619         }
620         // check enough target token (if target token is native) has been sent for the
            trade
621         if (_targetToken == NATIVE_TOKEN && msg.value < sourceAmount) {
622             revert InsufficientNativeTokenSent();
623         }
624         _targetToken.safeTransferFrom(msg.sender, address(this), sourceAmount);
625         // transfer the tokens to caller
626         token.safeTransfer(msg.sender, targetAmount);
627
628         // if the target token is native, refund any excess native token to caller
629         if (_targetToken == NATIVE_TOKEN && msg.value > sourceAmount) {
630             payable(msg.sender).sendValue(msg.value - sourceAmount);
631         }
632
633         // if no final target token is defined, transfer the target token to '
            transferAddress'
634         if (Token.unwrap(_finalTargetToken) == address(0)) {
635             // safe due to nonreentrant modifier (forwards all available gas if token is
                native)
636             _targetToken.unsafeTransfer(_transferAddress, sourceAmount);
637             // increment total collected in 'transferAddress'
638             _totalCollected += sourceAmount;
639         }
640
641         // if remaining balance is below the min token sale amount, reset the auction
642         if (_amountAvailableForTrading(token) < _minTokenSaleAmounts[token]) {
643             _resetTrading(token, 0);
644         }
645
646         return sourceAmount;
647     }

```

Listing 3.2: CarbonVortex::\_sellTokenForTargetToken()

**Recommendation** Revisit the above routine to follow the best practice of the checks-effects-interactions pattern. In the meantime, we acknowledge that the use of `nonReentrant` effectively blocks this specific risk.

**Status** The issue has been resolved by the following PR: 146.

### 3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [3]

#### Description

In the Carbon protocol, there is a special `admin` account (with the `ROLE_ADMIN`). This `admin` account plays a critical role in governing and regulating the protocol-wide operations (e.g., assign roles, configure parameters, withdraw funds, and upgrade the proxy). Our analysis shows that the `admin` account and other privileged roles need to be scrutinized. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

```

131     function setRewardsPPM(uint32 newRewardsPPM) external onlyAdmin validFee(
132         newRewardsPPM) {
133         _setRewardsPPM(newRewardsPPM);
134     }
135     ...
136     function setPriceResetMultiplier(
137         uint32 newPriceResetMultiplier
138     ) external onlyAdmin greaterThanZero(newPriceResetMultiplier) {
139         _setPriceResetMultiplier(newPriceResetMultiplier);
140     }
141     ...
142     function setMinTokenSaleAmountMultiplier(
143         uint32 newMinTokenSaleAmountMultiplier
144     ) external onlyAdmin greaterThanZero(newMinTokenSaleAmountMultiplier) {
145         _setMinTokenSaleAmountMultiplier(newMinTokenSaleAmountMultiplier);
146     }
147     ...
148     function setPriceDecayHalfLife(
149         uint32 newPriceDecayHalfLife
150     ) external onlyAdmin greaterThanZero(newPriceDecayHalfLife) {
151         _setPriceDecayHalfLife(newPriceDecayHalfLife);
152     }
153     ...
154     function setTargetTokenPriceDecayHalfLife(
155         uint32 newPriceDecayHalfLife
156     ) external onlyAdmin greaterThanZero(newPriceDecayHalfLife) {
157         _setTargetTokenPriceDecayHalfLife(newPriceDecayHalfLife);
158     }
159     ...
160     function setTargetTokenPriceDecayHalfLifeOnReset(
161         uint32 newPriceDecayHalfLife
162     ) external onlyAdmin greaterThanZero(newPriceDecayHalfLife) {

```

```

162     _setTargetTokenPriceDecayHalfLifeOnReset(newPriceDecayHalfLife);
163 }
164 ...
165 function setMaxTargetTokenSaleAmount(
166     uint128 newMaxTargetTokenSaleAmount
167 ) external onlyAdmin greaterThanZero(newMaxTargetTokenSaleAmount) {
168     _setMaxTargetTokenSaleAmount(newMaxTargetTokenSaleAmount);
169 }
170 ...
171 function setMinTargetTokenSaleAmount(
172     uint128 newMinTargetTokenSaleAmount
173 ) external onlyAdmin greaterThanZero(newMinTargetTokenSaleAmount) {
174     _setMinTokenSaleAmount(_targetToken, newMinTargetTokenSaleAmount);
175 }
176 ...
177 function disablePair(Token token, bool disabled) external onlyAdmin {
178     _setPairDisabled(token, disabled);
179 }
180 ...
181 function withdrawFunds(
182     Token[] calldata tokens,
183     address payable target,
184     uint256[] calldata amounts
185 ) external validAddress(target) validateTokens(tokens) nonReentrant onlyAdmin {
186     uint256 len = tokens.length;
187     if (len != amounts.length) {
188         revert InvalidAmountLength();
189     }
190     for (uint256 i = 0; i < len; i = uncheckedInc(i)) {
191         // safe due to nonReentrant modifier (forwards all available gas in case of
192         // ETH)
193         tokens[i].unsafeTransfer(target, amounts[i]);
194     }
195     emit FundsWithdrawn({ tokens: tokens, caller: msg.sender, target: target,
196         amounts: amounts });
197 }

```

Listing 3.3: Example Privileged Operations in CarbonController

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

In the meantime, the above contract makes use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

**Recommendation** Promptly transfer the privileged accounts to the intended DAO-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and en-

sure the intended trustless nature and high-quality distributed governance.

**Status** The issue has been mitigated as the team confirms they will use a multi-sig account as the admin.





## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `CarbonVortex` protocol. Note `Carbon` is a next-generation AMM which introduces the key new feature of asymmetric liquidity, wherein any given bonding curve only trades in a single direction, effectively being an out-of-the-money limit order. And `CarbonVortex` collects fees from trades performed on the `Carbon DeFi` network, and as a result of arbitrage transactions performed using the `Bancor Fastlane`. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



## References

- [1] MITRE. CWE-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-190: Integer Overflow or Wraparound. <https://cwe.mitre.org/data/definitions/190.html>.
- [3] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Numeric Errors. <https://cwe.mitre.org/data/definitions/189.html>.
- [7] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [8] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.

- [11] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

